# *Map Reduce*

# Map Reduce

- Not a data model, but a processing/programming model
- Focus: the abstraction/model and how to use it
- Original Google paper
  - http://research.google.com/archive/mapreduce.html
- A great textbook available on-line
  - http://lintool.github.io/MapReduceAlgorithms/ed1n.html

# Map Reduce

- Programming model introduced by Google (2004)
- Not really a new concept
  - Dates back to ideas from functional programming
- Very useful for processing large datasets
- Infrastructure
  - Hadoop, Amazon Elastic Map Reduce,…
- Also treated as a programming pattern
  - MongoDB and other systems support it

# What is Map Reduce?

- A programming model and infrastructure for parallel programming
- Doing very large-scale data processing requires parallelization

# *What is Map Reduce?*

- Not trivial to parallelize arbitrary task:
  - What are the subproblems?
  - How do we get the data to/from the workers working on each subproblem?
  - How do we synchronize and share results as needed between workers?

# *Example*

- You get a job at Google
- Your boss says: hey, we have a large corpus of documents
  - All the pages on the Web
- We need some statistics - a list of word frequency occurrences across the whole corpus
- We need the results fast, so don't do it all on one machine.
- What do you do?

# *Typical Workflow*

- Iterate over a large number of records
- *Extract something of interest from each*
- Bring together intermediate results
- *Aggregate intermediate results*
- Generate final output

- Most of the real "computation" occurs in the two blue phases

# Map Reduce

- (Or map-reduce, mapreduce, etc.)
- A general framework for writing parallel programs that follow the workflow we saw
- Idea:
  - You write the code for the two blue phases
    - Because that's what is unique to your computation
  - System takes care of the rest

# Typical Workflow

- Iterate over a large number of records
- Map: Extract something of interest from each
- Bring together intermediate results
  - In some standardized way
- Reduce: Aggregate intermediate results
- Generate final output

# *Understanding Map Reduce*

- Key to understanding Map Reduce is the third point in workflow
  - How are intermediate results brought together?
  - The framework does it for you, so you have to understand how it does it

# *Understanding Map Reduce*

- Fundamental idea: key-value model
  - This is the data model for Map Reduce jobs
  - Helps provide a unified interface for bringing results together

# *MR and key-value model*

- Key-value is simple data model
- Map-Reduce uses that as input and output format
- Map function takes one key-value pair as input and outputs a set of key-value pairs
  - The input and output keys can be different

# *Example: Word Count*

```
map(String key, String value):
    // key: document id
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");
```

A mapper utility can apply this in parallel to a whole lot of documents

# *Now what?*

- Have a whole lot of key-value pairs after the map
- Need to bring them together and aggregate
- Would like to bucketize/ GROUP BY something so can compute in parallel again
  - What to bucketize by?
    - The English word (i.e. the output key of the mapper)

# Example: Word Count

```
reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(key, AsString(result));
```

Can also run this in parallel

# *So what do we have so far?*

- See how to write map and reduce functions
  - Work on a key-value model
- Believe that both map and reduce functions can be executed in parallel
- But what about the middle step?
  - Bucketize output of mapper based on value of output key…
- Fortunately, the exact same middle step is useful for a lot of other problems
  - So map reduce frameworks have it built-in!
  - You don't need to write this logic yourself

# *Map Reduce*

- You only need to specify two functions:

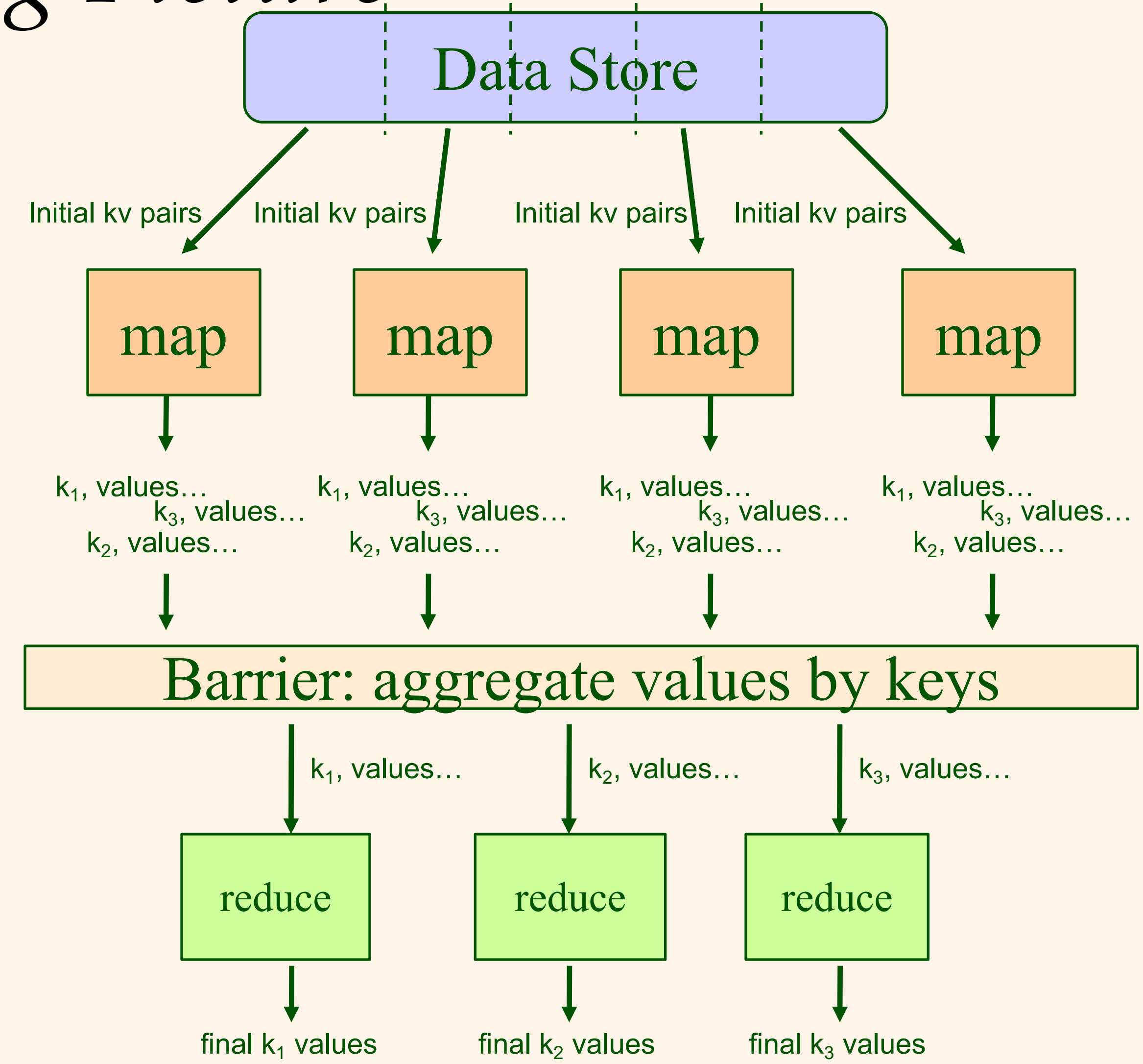
  **map** (k, v) → [(k', v')]
  **reduce** (k', [v']) → [(k'', v'')]

  - [ … ] denotes a list

# Map Reduce

- Framework takes care of the actual execution:
  - Applies your map function to every initial (k,v)
  - Reshuffles the output of the map to group by k'
  - Applies your reduce function

# The Big Picture



# Example: Word Count

```
map(String key, String value):
     // key: document id
     // value: document contents
     for each word w in value:
          EmitIntermediate(w, "1");

reduce(String key, Iterator values):
     // key: a word
     // values: a list of counts
     int result = 0;
     for each v in values:
          result += ParseInt(v);
     Emit(key, AsString(result));
```

# Map and Reduce Functions

- Do not have to be purely functional
- Can keep internal state across multiple inputs
- Can also have external side effects
  - E.g. write to files
- Should be careful about using external resources
  - E.g. if have multiple mappers and/or reducers contending for same database, could become a bottleneck

# *Map and Reduce Functions*

- Possible to have programs <u>without a reduce</u>
  - Mappers just apply some computation in parallel to a dataset
- Impossible to have programs <u>without a map</u>
  - But map could be identity function
  - Use framework to re-sort and re-group key-value pairs before feeding to reducers
- Could have reducer identity function too
  - Computation occurs in map phase
  - Framework used to re-sort and re-group output of mapper
- Could even have both mapper and reducer as identity functions

# *Map Reduce Frameworks*

- Various frameworks to support this programming pattern
  - Google's own internal implementation, Hadoop, Amazon EMR
  - Also supported in MongoDB

# *Map Reduce Frameworks*

- Implementations vary a bit in what exact functionality they allow/support
  - E.g. whether reducer input and output keys must be the same
  - Or whether they support additional functions like partitioners and combiners
  - Or what happens in corner cases (e.g. MongoDB won't run reducer if there is only one value for a reducer input key)

# *Programming in Map Reduce*

- Powerful abstraction, but requires different way of thinking about programming
- Things to figure out:
  - How to impose key-value structure on problem?
  - What can be parallelized?
  - How to deal with the fact that there is no "global state" anymore (or at least that you should avoid using global state?)
- Next: a few examples to get you more comfortable with doing things in MR

# *Inverted Index*

- How to use Map Reduce to build an inverted index?
- Given a set of documents as input
- Want as output a set of entries of the form:

  <*word, list of documents it appears in*>
- May assume documents have unique ids
- Very useful in practice e.g. in search engines

# Inverted Index

```
map(String docid, String contents):
    for each word w in contents:
        EmitIntermediate(w, docid);

reduce(String word, Iterator docids):
    List result = new ArrayList();
    for each d in docids:
        result.add(d);
    Emit(word, result);
```

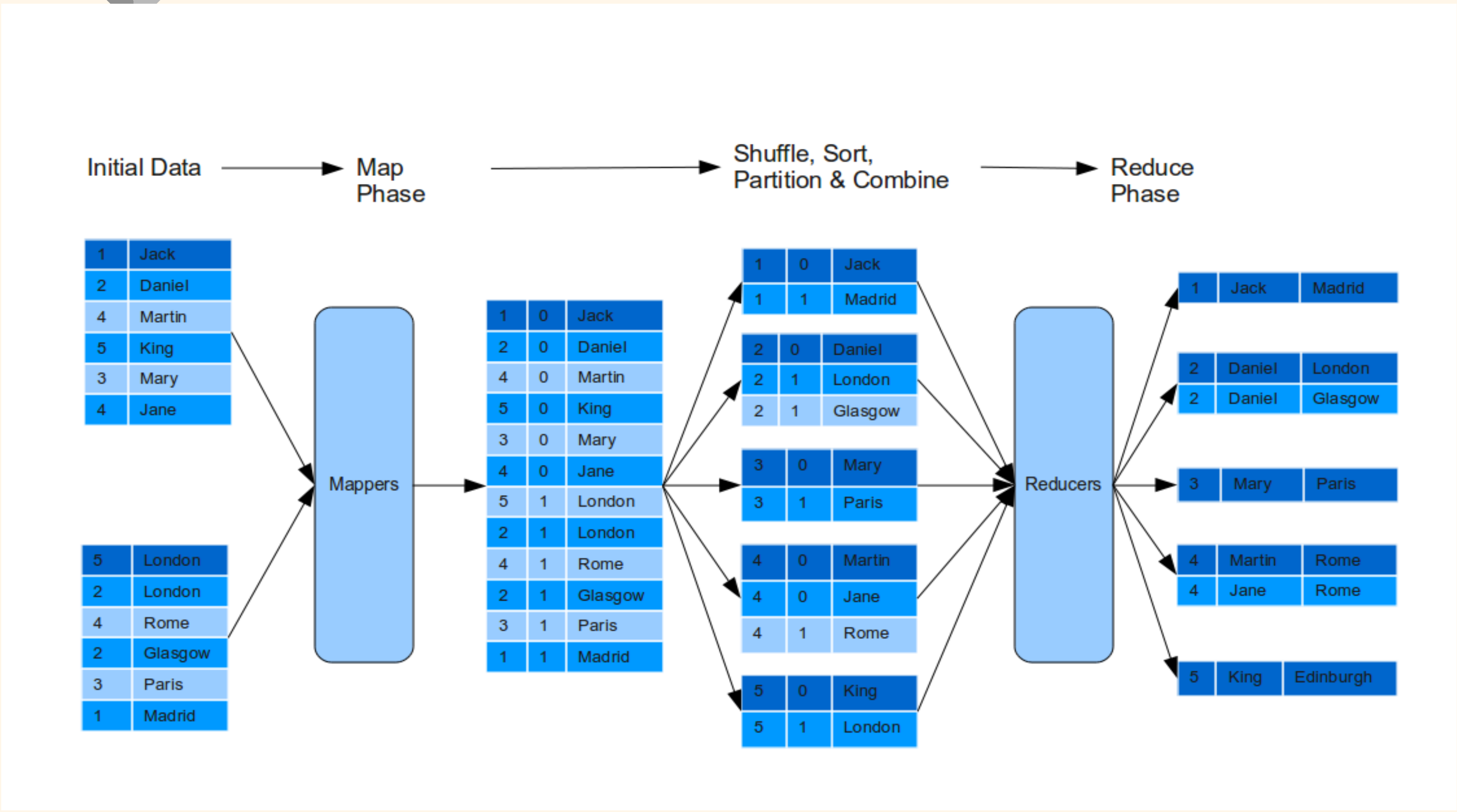
# *Inverted Index*

- Easy to modify this to make it more fancy
  - Keep track of word positions within documents
  - Etc…

# *Example 2: Joins*

- Have two large datasets representing two large relations
  - In some reasonable format, e.g. with one line per row
- Need to peform a join on a particular column
- How to do it with Map Reduce??

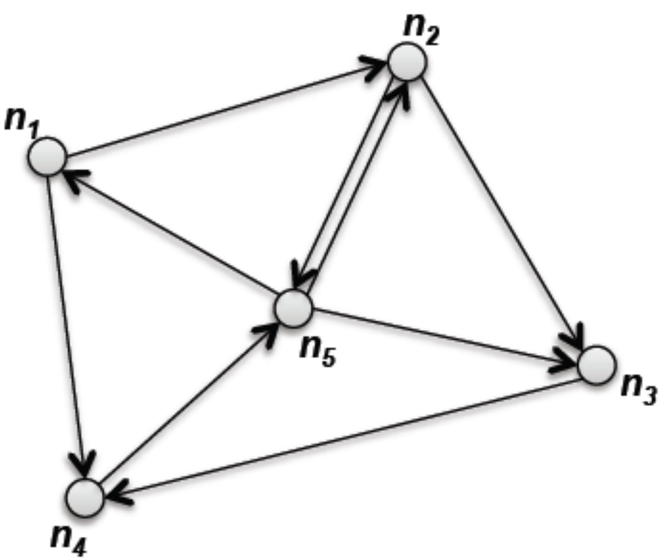
# Example: Join



# *Iterative Map Reduce*

- You are not limited to a single MR run
- Output of one MR job can serve as input to the next one
  - MR pipelines
  - Need to make sure your formats are compatible!
- Iterative algorithms also possible and useful
- First case study: graph processing

# *Representing Graphs*




|  | $n_1$ | $n_2$ | $n_3$ | $n_4$ | $n_5$ |
|---|---|---|---|---|---|
| $n_1$ | 0 | 1 | 0 | 1 | 0 |
| $n_2$ | 0 | 0 | 1 | 0 | 1 |
| $n_3$ | 0 | 0 | 0 | 1 | 0 |
| $n_4$ | 0 | 0 | 0 | 0 | 1 |
| $n_5$ | 1 | 1 | 1 | 0 | 0 |

adjacency matrix

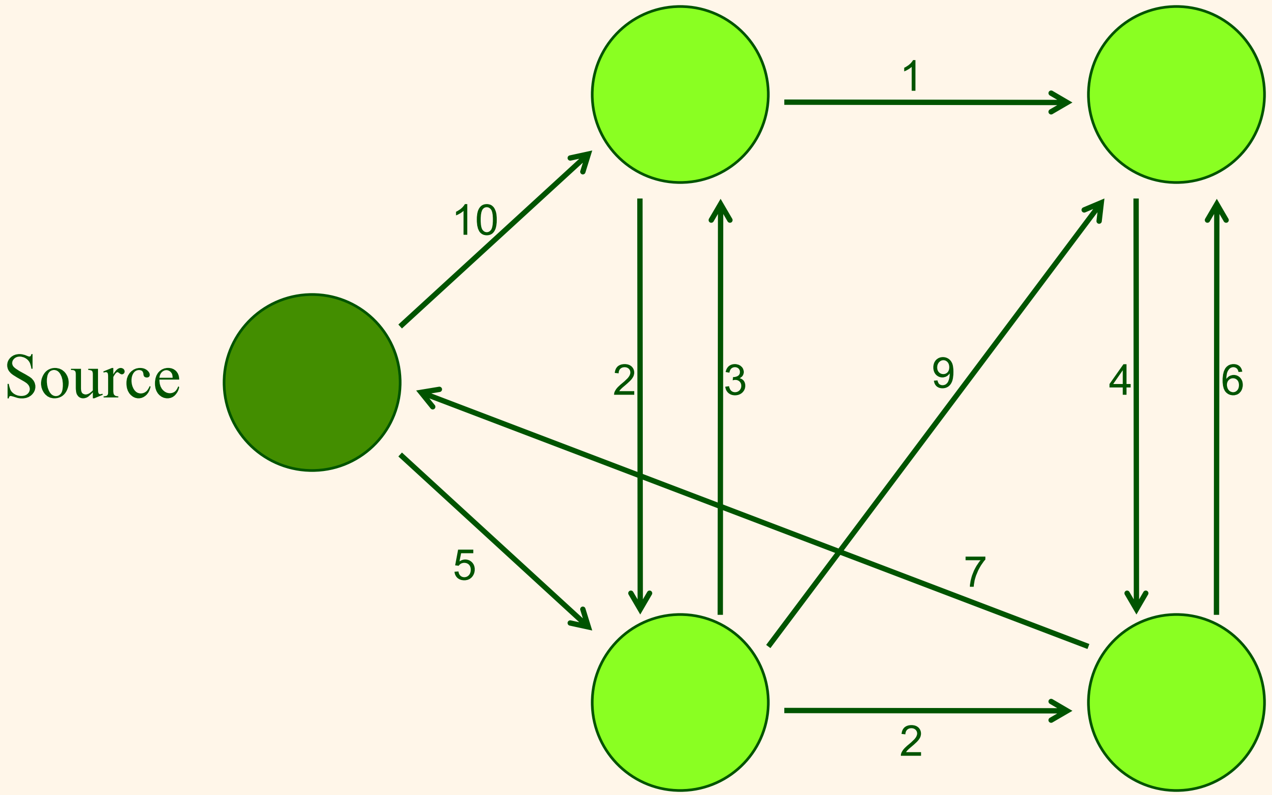
$n_1$ [$n_2$, $n_4$]
$n_2$ [$n_3$, $n_5$]
$n_3$ [$n_4$]
$n_4$ [$n_5$]
$n_5$ [$n_1$, $n_2$, $n_3$]

adjacency lists

# *Single Source Shortest Path*

- **Problem:** find shortest path from a source node to one or more target nodes
- Single processor machine: Dijkstra's Algorithm
- MapReduce: parallel Breadth-First Search (BFS)

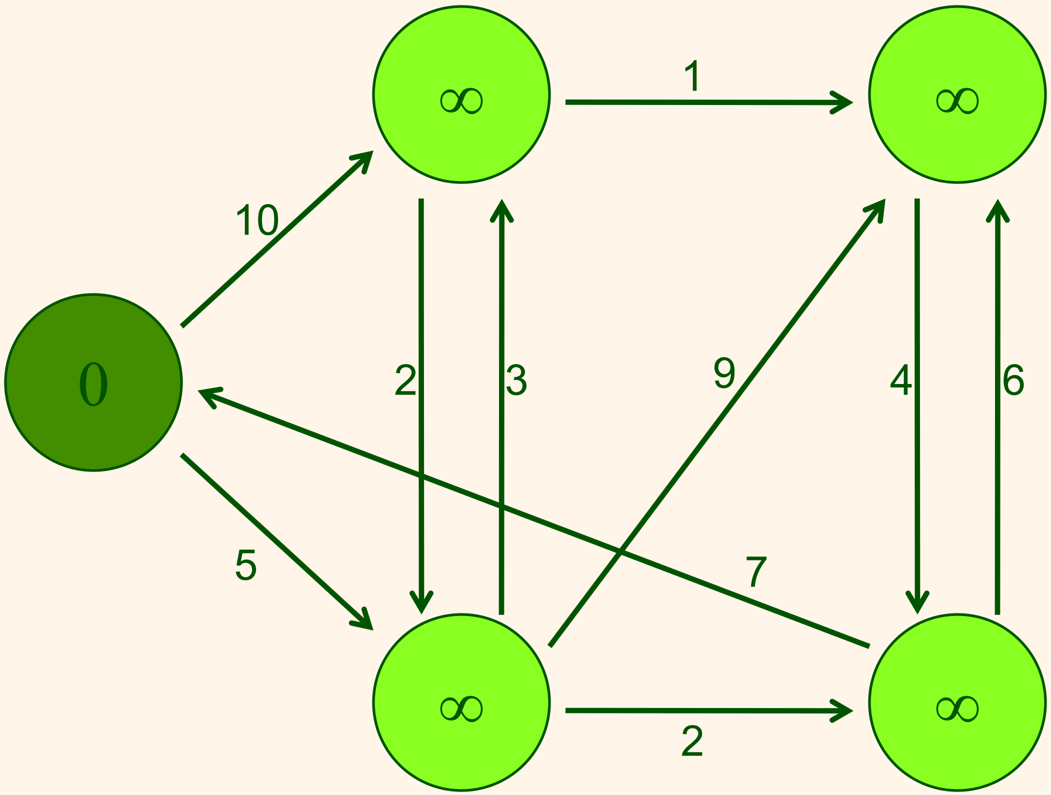
# SSSP



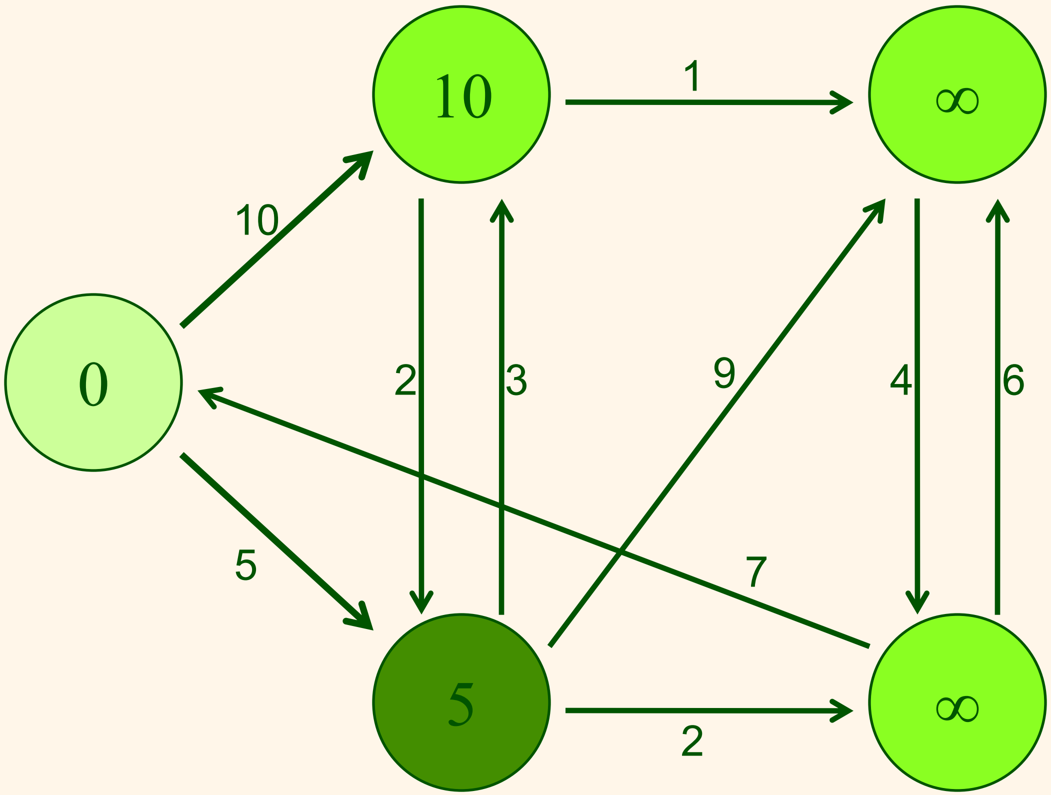
# *Dijkstra's Algorithm Review*

$\textsc{Dijkstra}(G, w, s)$
  $d[s] \leftarrow 0$
  **for all** vertex $v \in V$ **do**
    $d[v] \leftarrow \infty$
  $Q \leftarrow \{V\}$
  **while** $Q \neq \emptyset$ **do**
    $u \leftarrow \textsc{ExtractMin}(Q)$
    **for all** vertex $v \in u.\textsc{AdjacencyList}$ **do**
      **if** $d[v] > d[u] + w(u, v)$ **then**
        $d[v] \leftarrow d[u] + w(u, v)$

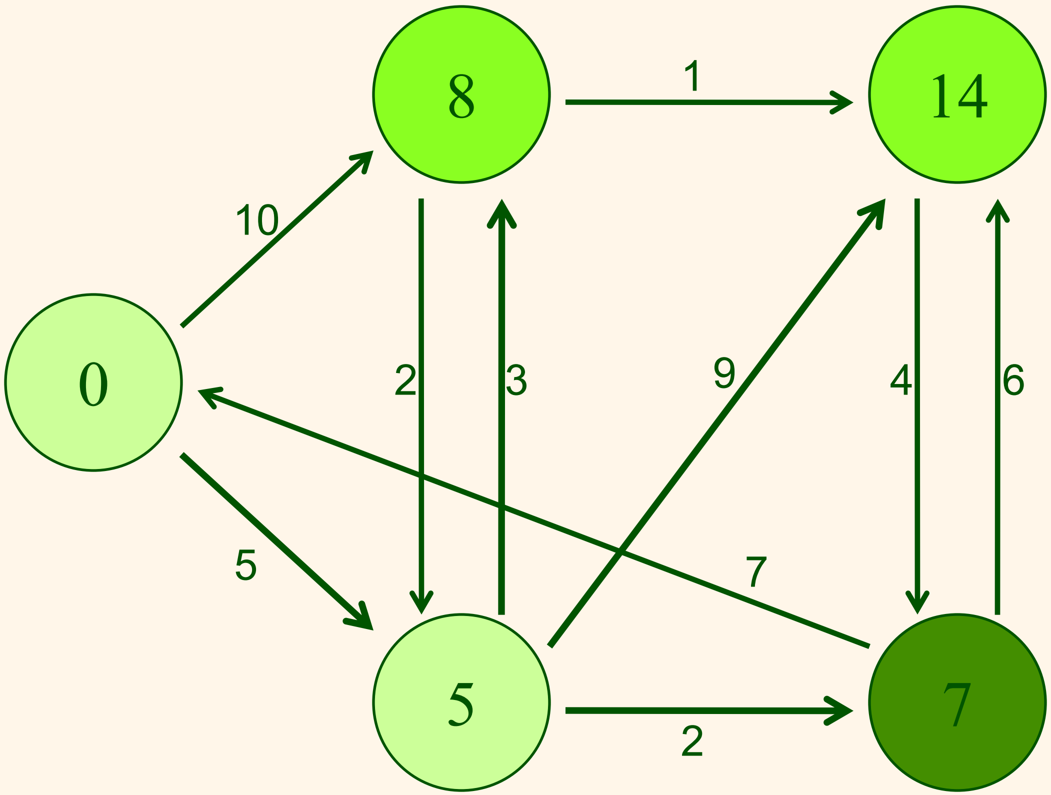
# Dijkstra's Algorithm Example



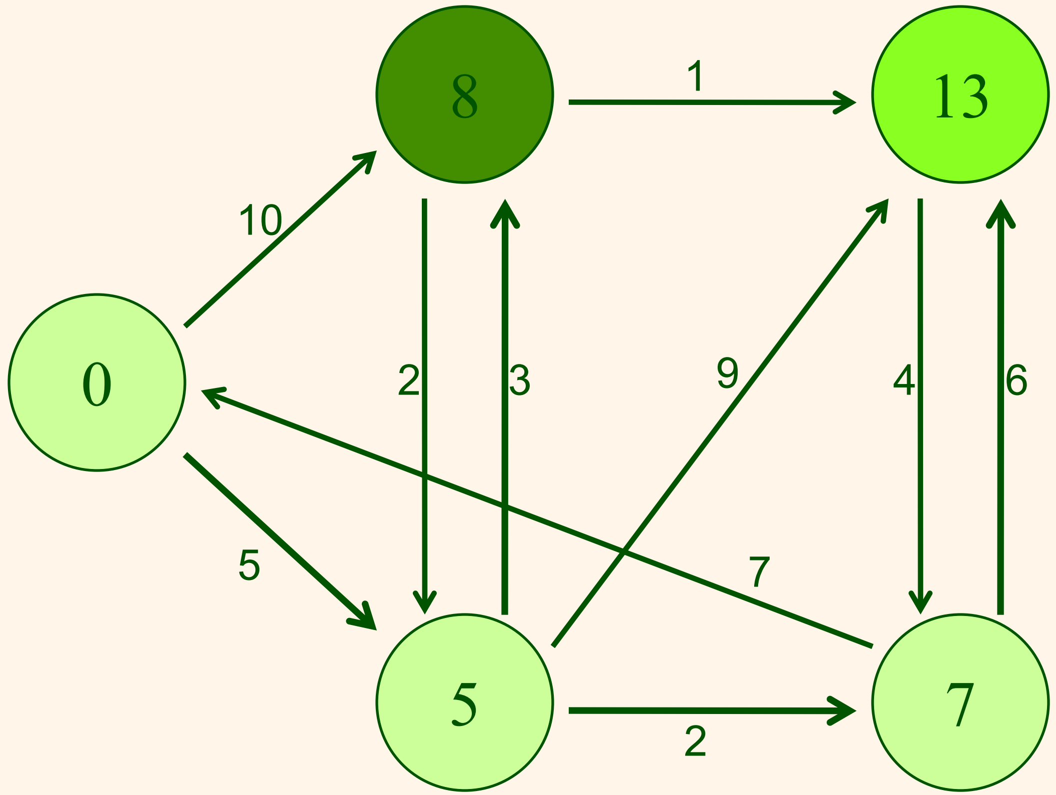
# Dijkstra's Algorithm Example



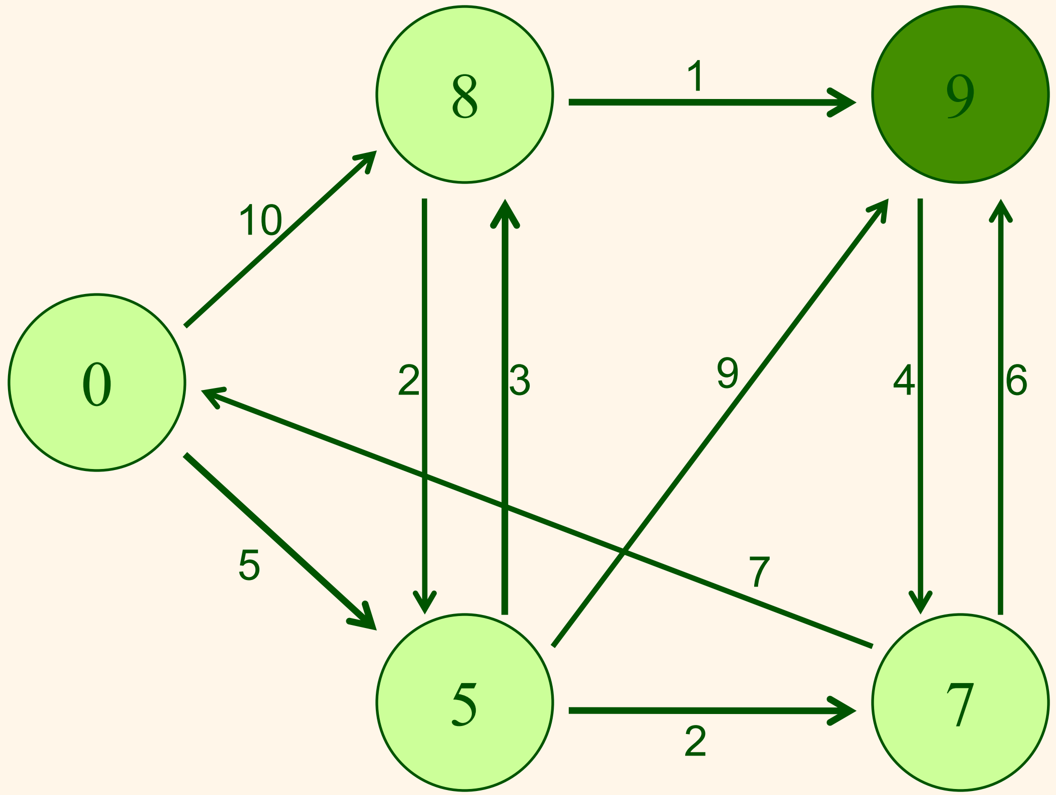
# Dijkstra's Algorithm Example



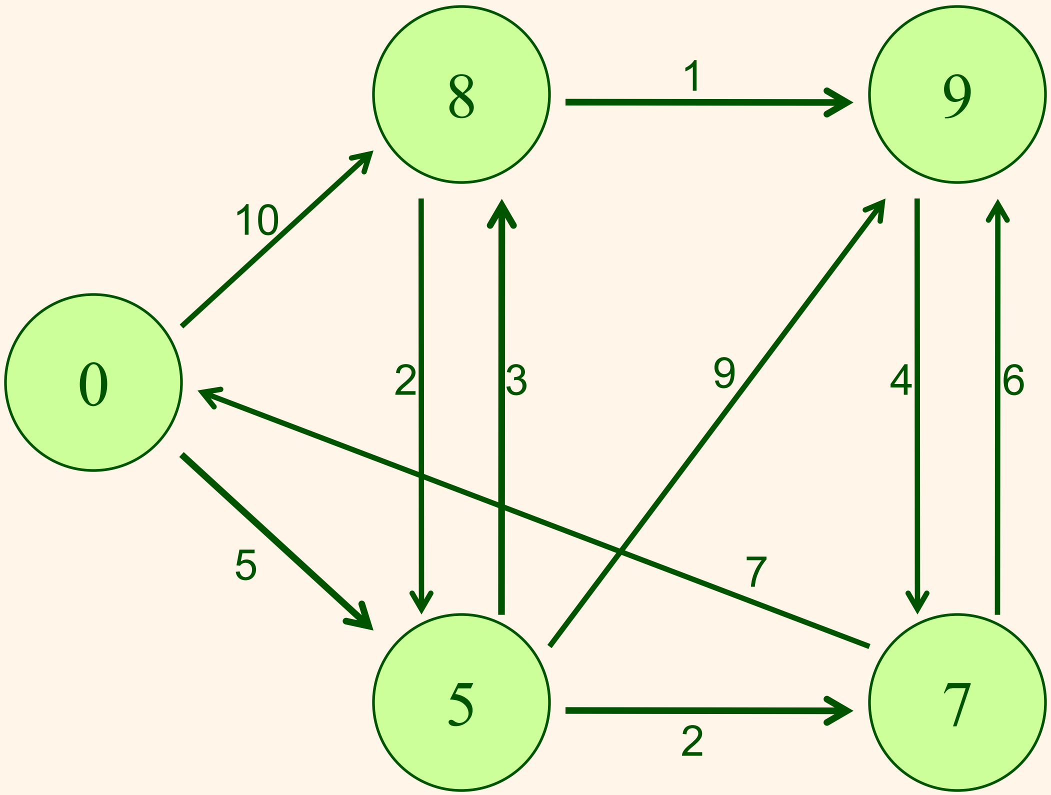
# Dijkstra's Algorithm Example



# Dijkstra's Algorithm Example



# Dijkstra's Algorithm Example



# *What about Map Reduce?*

- Suppose we have a very, very, very big graph
- And would like to compute this information quickly and in parallel
  - Using the magic of Map Reduce
- Definitely can't run Dijkstra's algorithm directly
  - Can't have a global queue!

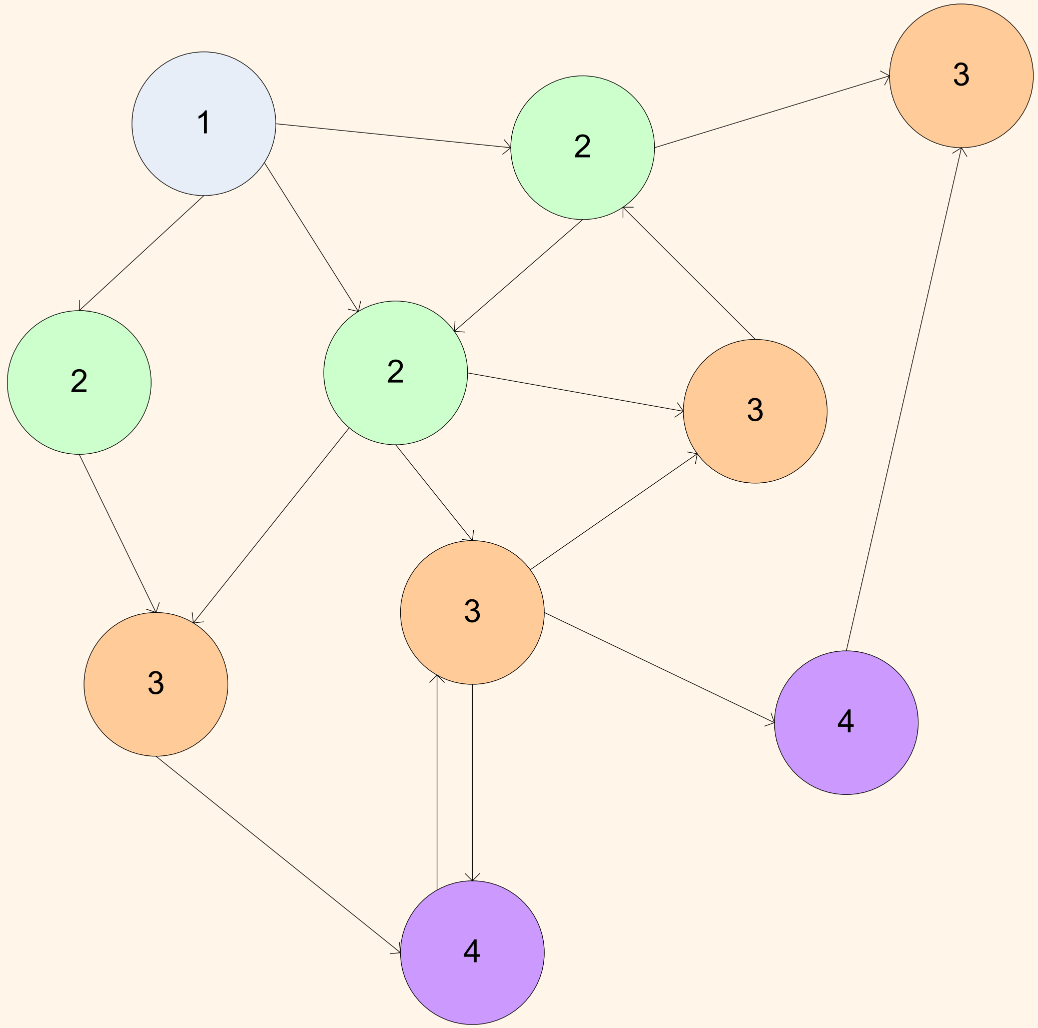
# SSSP in Map Reduce

- Can't run Dijkstra's algorithm directly
  - Can't have a global queue!
- Another way to do it: Parallel BFS
- Start by assuming all edge weights are equal
  - Will relax this later

# *Finding the Shortest Path*

- Intuition: process all nodes at each step
- Some nodes have no information (distance = infinity)
  - So can't do much
- But other nodes do know something
  - E.g. source knows it is at distance 0
  - So can pass this fact on to its out-neighbors
    - Who now know they are at distance 1!
  - At the next iteration, these neighbors know they're at distance 1
    - So can tell *their* out-neighbors they're at distance 2.

# Parallel BFS



# *From Intuition to Algorithm*

- A map task receives
  - Key: node $n$
  - Value: D (distance from start), points-to (list of nodes reachable from $n$)
- $\forall p \in$ points-to: emit ($p$, D+1)
- The reduce task gathers possible distances to a given $p$ and selects the minimum one

- Possible through the magic of the "sort and shuffle" between Map and Reduce
  - Map processes node and updates distances of out-neighbors
  - Reduce processes node based on info from its in-neighbors

# *Multiple Iterations Needed*

- Each Map Reduce task advances the "known frontier" by one hop
  - Subsequent iterations include more reachable nodes as frontier advances
  - Multiple iterations are needed to explore entire graph
  - Feed output back into the same MapReduce task

# *Multiple Iterations Needed*

- Passing along the graph structure:
  - Next iteration of Map  needs points-to list again
  - So need to "carry" it with us as we run the algorithm

# *Map*

```
class MAPPER
    method MAP(nid n, node N)
        d ← N.DISTANCE
        EMIT(nid n, N)
        for all nodeid m ∈ N.ADJACENCYLIST do
            EMIT(nid m, d + 1)
```

# *Reduce*

**class** REDUCER
  **method** REDUCE(nid $m$, $[d_1, d_2, \ldots]$)
    $d_{min} \leftarrow \infty$
    $M \leftarrow \emptyset$
    **for all** $d \in$ counts $[d_1, d_2, \ldots]$ **do**
      **if** ISNODE($d$) **then**
        $M \leftarrow d$
      **else if** $d < d_{min}$ **then**
        $d_{min} \leftarrow d$
    $M$.DISTANCE $\leftarrow d_{min}$
    EMIT(nid $m$, node $M$)

# Termination

- Eventually, all nodes will be discovered, all edges will be considered (in a connected graph)
- Stop when there are no nodes with a distance of infinity
  - Can be checked by the driver/harness/program that runs the outer loop and schedules each Map Reduce job

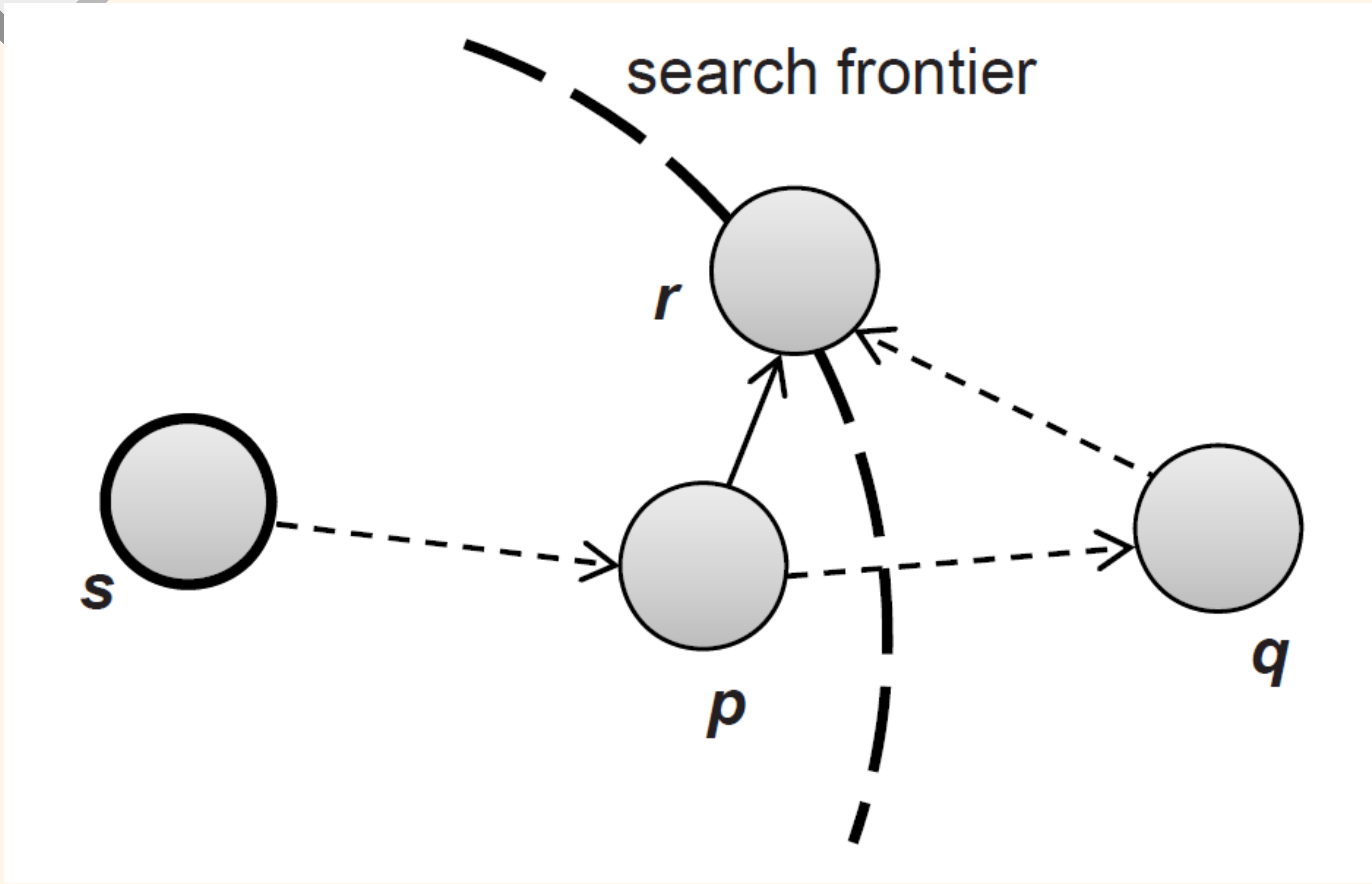
# *Weighted Edges*

- Now add positive weights to the edges

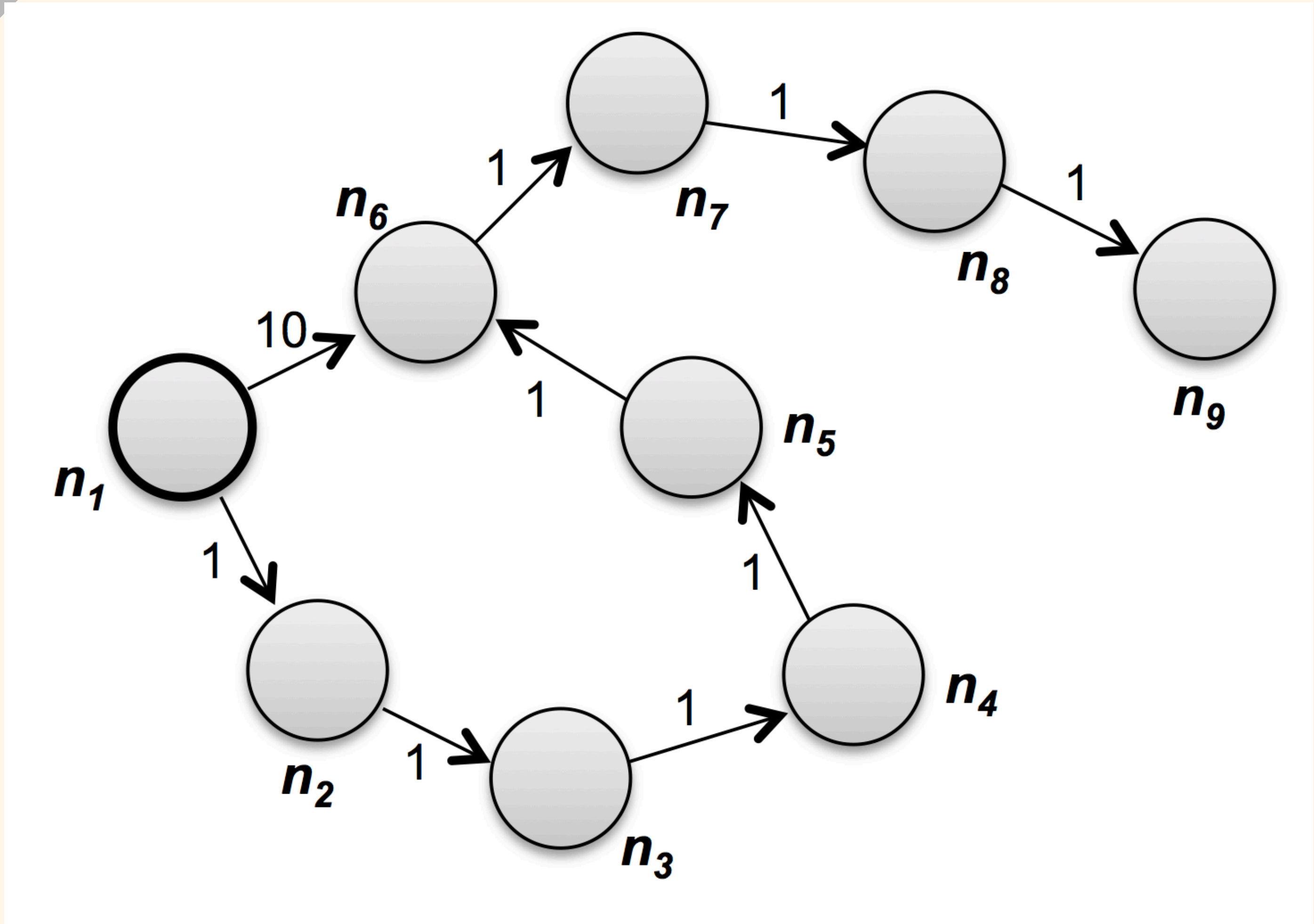
- Simple change: points-to list in map task includes a weight $w$ for each pointed-to node
  - emit ($p$, D+$w_p$) instead of ($p$, D+1) for each node $p$

- Termination behavior different
  - Just because we've reached a node doesn't mean we've found the shortest path to it!

# Node Exploration Process



# Node Exploration Process



# *Termination*

- When distances have not changed during an iteration, safe to stop

# Comparison to Dijkstra

- Dijkstra's algorithm is more efficient
  - At any step it only pursues edges from the minimum-cost path inside the frontier
  - Only processes each node once
- MapReduce explores all paths in parallel
  - Does a lot of recomputation
    - Not a bug, need it to handle situations where the "shortest" path contains more edges than another available path
  - But can be done in parallel

# *General Approach*

- Graph algorithms with MapReduce:
  - Each map task receives a node and its outlinks
  - Map task compute some function of the link structure, emits value with target as the key
  - Reduce task collects keys (target nodes) and aggregates
- Iterate multiple MapReduce cycles until some termination condition

# PageRank

- Google's famous algorithm for ranking Web Pages
  - A measure of "quality reputation" of a page
  - Useful for ranking/ordering search results

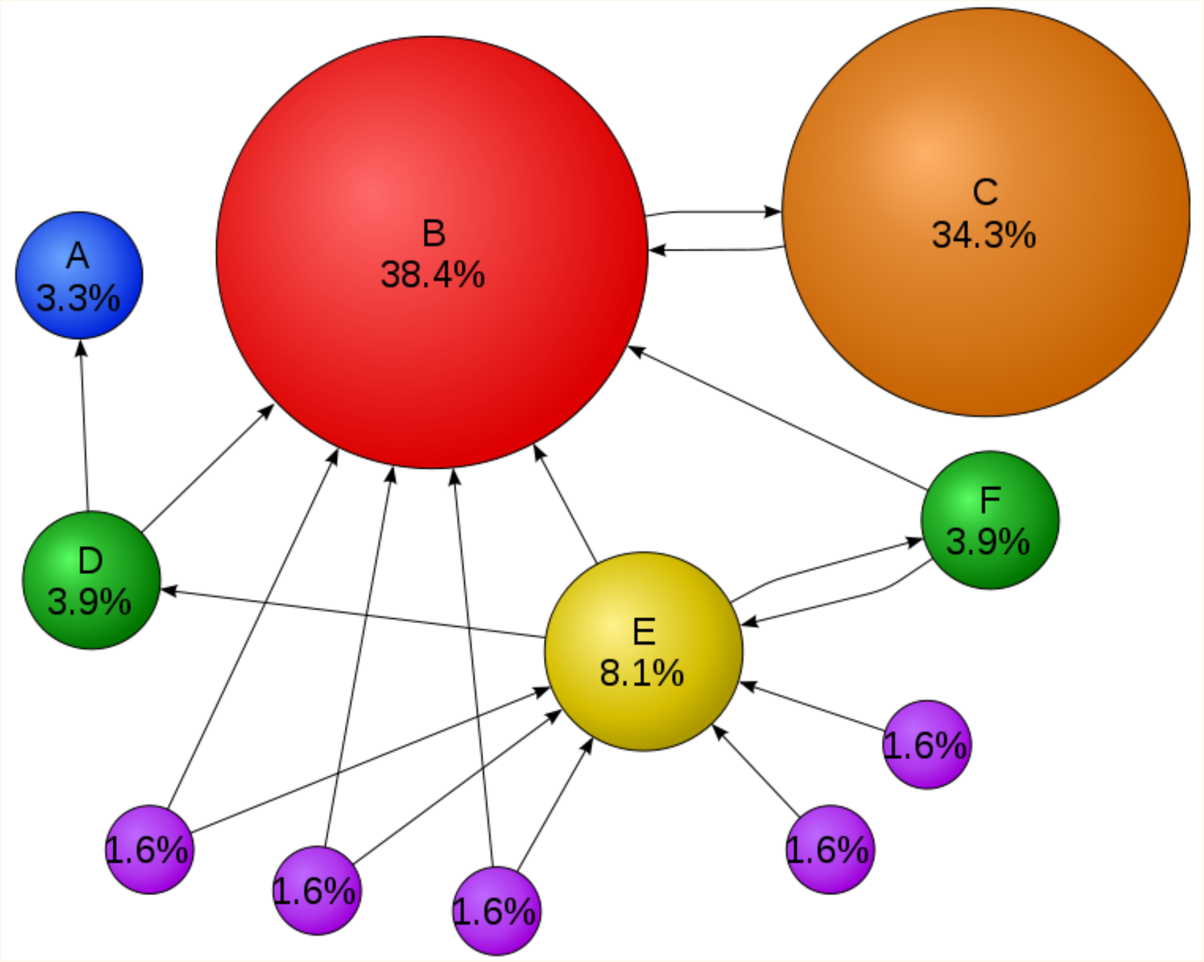
# Random Surfer Model

- Intuition for PageRank
- Imagine a surfer who starts on a randomly chosen page and then follows outgoing links at random
  - Markov process
- PageRank is probability that user will arrive at a given page during this random walk

# *A little more complex!*

- Model assumes that surfer doesn't always follow a link, but sometimes e.g. bookmarks instead.
- Before each move, surfer flips a coin
  - With probability $1 - \alpha$ , follows an out-link
  - With probability $\alpha$ , teleports to a (uniformly chosen) random page

# PageRank Example

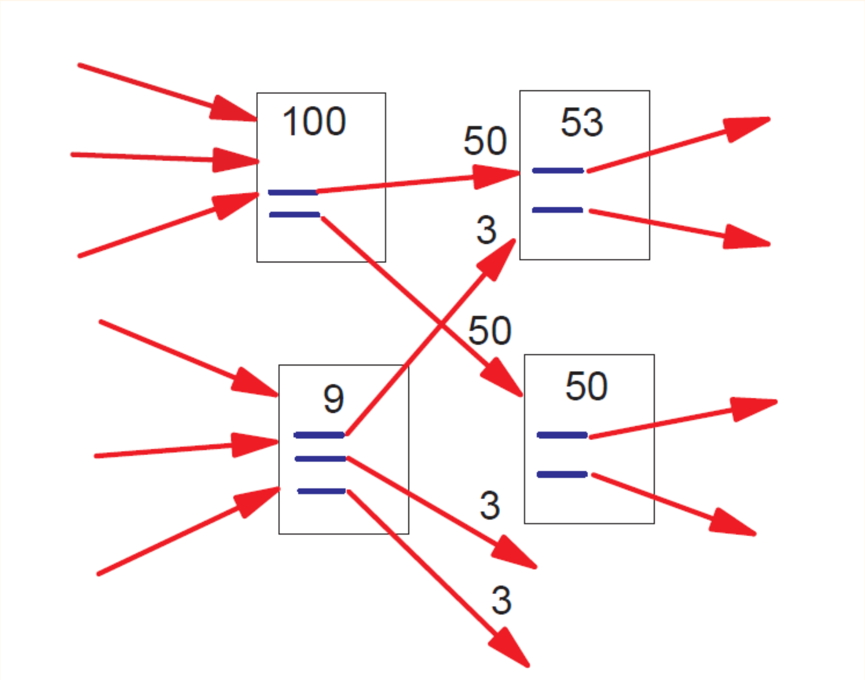


# Computing PageRank

- Let's start with some simplifying assumptions
  - Assume all nodes have at least one outlink
  - And surfer never does a random restart using bookmarks
- Will talk about how to lift these assumptions soon

# Simplified PageRank Intuition

- PageRank of a page is based on the PageRank of the pages which link to it
- A page divides its PageRank equally among all its outgoing links



# *Somewhat more formally*

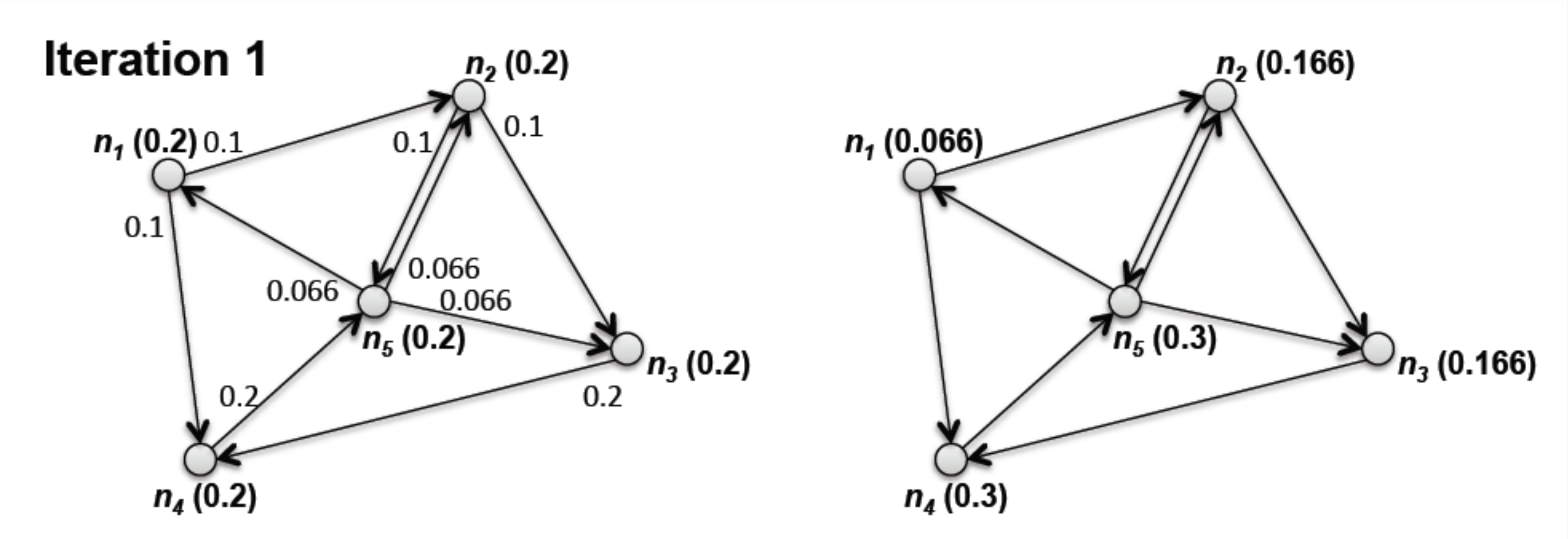
$$P(n) = \Sigma_{m \in L(n)} \frac{P(m)}{C(m)}$$

- L(n) is the set of pages that link to n and C(m) is the number of out-neigbors of page m

# *Iterative computation of SPR*

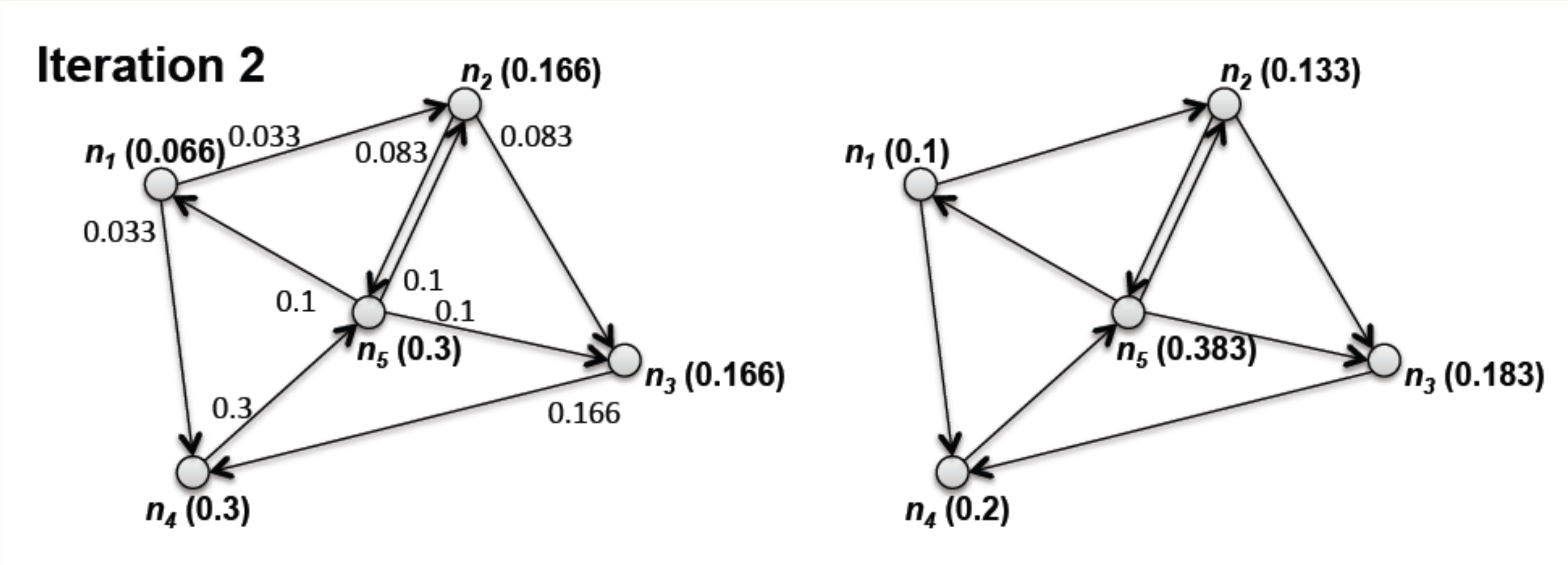
- How do we compute this?
- Various methods, but we are interested in Map Reduce
- Idea:
  - Initialize everything to the same PageRank (1/number of nodes)
  - "pass around" PageRank contributions from nodes to their out-neighbors

# Example



Iteration 1
$n_1$ (0.2)
$n_2$ (0.2)
$n_3$ (0.2)
$n_4$ (0.2)
$n_5$ (0.2)
0.1
0.1
0.1
0.1
0.066
0.066
0.066
0.2
0.2
$n_1$ (0.066)
$n_2$ (0.166)
$n_3$ (0.166)
$n_4$ (0.3)
$n_5$ (0.3)

# Example



Iteration 2
$n_2$ (0.166)
$n_1$ (0.066)
0.033
0.083
0.083
0.033
0.1
0.1
0.1
$n_5$ (0.3)
$n_3$ (0.166)
0.3
0.166
$n_4$ (0.3)
$n_2$ (0.133)
$n_1$ (0.1)
$n_5$ (0.383)
$n_3$ (0.183)
$n_4$ (0.2)

# Map

```
class Mapper
    method Map(nid n, node N)
        p ← N.PageRank/|N.AdjacencyList|
        Emit(nid n, N)                              ▷ Pass along graph structure
        for all nodeid m ∈ N.AdjacencyList do
            Emit(nid m, p)                          ▷ Pass PageRank mass to neighbors
```

# Reduce

```
class Reducer
    method Reduce(nid m, [p1, p2, ...])
        M ← ∅
        for all p ∈ counts [p1, p2, ...] do
            if IsNode(p) then
                M ← p                      ▷ Recover graph structure
            else
                s ← s + p                  ▷ Sum incoming PageRank contributions
        M.PageRank ← s
        Emit(nid m, node M)
```

# *Iterate*

- Full algorithm is iterative
- Initialize the nodes to uniform distribution
- Run the two MR jobs described iteratively
- Until convergence (no change)

# *Now, back to dangling nodes*

- If any PageRank is lost due to nodes with no out-neighbors, redistribute that PageRank uniformly throughout the graph for next iteration
- In the Map Reduce model: keep track of any lost PageRank
  - E.g. by using a special reserved intermediate key, or using a counter (i.e. storing it somewhere)

# *Solution*

- After the MR task is done, do a cleanup pass
- Deal both with "missing mass" and with random restart factor

# *Cleanup pass*

- Adjust the PageRank of each node to be

$$p' = \alpha \left( \frac{1}{|G|} \right) + (1 - \alpha) \left( \frac{m}{|G|} + p \right)$$

- Where *p* is the current PageRank, *m* is the mass lost due to sinks, and $|G|$ is the number of nodes in the graph
- This can be done using a map job (no reduce)

# *The full PageRank Algorithm*

- Initialize the nodes to uniform distribution
- Run the two MR jobs described iteratively
- Until convergence (no change)